AFRL-RI-RS-TR-2016-096

VERIFICATION GAMES: CROWD-SOURCED FORMAL VERIFICATION

UNIVERSITY OF WASHINGTON

MARCH 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-096 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
DILIA E. RODRIGUEZ
Work Unit Manager

/ S /
RICHARD MICHALAK
Acting Technical Advisor, Computing & Communication Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAR 2016	FINAL TECHNICAL REPORT	JUN 2012 – SEP 2015

4. TITLE AND SUBTITLE

VERIFICATION GAMES: CROWD-SOURCED FORMAL VERIFICATION

5a. CONTRACT NUMBER
FA8750-12-C-0174

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Michael Ernst

5d. PROJECT NUMBER
EF00

5e. TASK NUMBER
78

5f. WORK UNIT NUMBER
80

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Washington
Office of Sponsored Programs
4333 Brooklyn Ave., N.E.
Seattle, WA 98195-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-096

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Over the more than three years of the project Verification Games: Crowd-sourced Formal Verification the verification tools developed by the Programming Languages and Software Engineering group were improved. A series of games were developed by the Center for Game Science: Pipe Jam, Traffic Jam, Flow Jam and Paradox. Verification tools and games were integrated to verify that the Hadoop-common program satisfies constraints that render it free of the following vulnerabilities: injection attacks, incorrect use of format strings, violations of documented locking conventions.

15. SUBJECT TERMS

type theory, games, Java code vulnerabilities, crowd-sourced formal verification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			DILIA E. RODRIGUEZ
U	U	U	SAR	21	19b. TELEPHONE NUMBER (*Include area code*) N/A

Contents

List of Figures

1 SUMMARY

Our increasing dependence on software makes it imperative to find more effective and efficient mechanisms for improving software reliability. Formal verification is an important part of this effort, since it is the only way to be certain that a given piece of software is free of (certain types of) errors. To date, formal verification has been done manually by specially-trained engineers. Labor costs have heretofore made formal verification too costly to apply beyond small, critical software components. However, if we were able to transform ordinary computer users into ones capable of performing verification tasks, we could achieve a dramatic reduction in the cost of producing verified code. The goal of CSFV (Crowd-Sourced Formal Verification) was to make verification more cost-effective by reducing the skill set required for program verification and increasing the pool of people capable of performing program verification. Our approach was to transform the verification task (a program and a goal property) into a visual puzzle task -- a game -- that is solved by non-experts. The solution of the puzzle is then translated back into a proof of correctness.

Over the course of the program, we developed and improved the code verification tools from the University of Washington Computer Science and Engineering's Programming Languages and Software Engineering group (PLSE), and combined them with games designed and implemented by the UW's Center for Game Science (CGS). We designed a series of games successively titled *Pipe Jam*, *Traffic Jam*, *Flow Jam*, and *Paradox* that present real verification problems as puzzles to players with no technical background. A game level in these games can also be thought of as a set of constraints that a player is trying to solve. Like many puzzle games, in order to complete a game level in this family of games, the player must find consistent settings for all the game elements. By the end of the program, over 7,000 unique players had played *Flow Jam* or *Paradox* for a combined total of over 7,500 hours of play, resulting in over 50,000 level solution submissions.

We made a number of concrete proofs on Hadoop, a large piece of a widespread and currently-used program in support of developing and testing our approach:

- We proved that the Hadoop-common program (100K non-comment, non-blank lines of Java code) has no operating system command injection attacks.
- We proved that Hadoop-common uses format strings correctly.
- We proved that Hadoop-common does not violate its documented locking conventions.
- We re-proved that the Hadoop-common program has no operating system command injection attacks, this time using type inference, eliminating the need for human intervention.

Finally, we performed an experiment to compare the cost of two techniques for verifying a program's correctness. One technique was the traditional one in which a human verification expert writes the specifications and then those specifications are automatically verified. The other technique was our crowd-sourced workflow where the specifications are inferred via gameplay, tweaked as needed by the verification expert, and then automatically verified. Our goal was to reduce the overall cost rather than to completely eliminate the human expert's job, a goal we believe is impractical at the current state of the art. Of the two conditions, unannotated code required 45 minutes total time (7 minutes of type checking and 38 minutes of manual effort) versus 4 minutes total when starting with game results (3 minutes of type-checking and 1 minute of manual effort). Not included in these timing were the annotation of APIs (determining the proof goal, required in both cases), and gameplay (crowd time, machine time to generate levels).

2 INTRODUCTION

Our increasing dependence on software makes it imperative to find more effective and efficient mechanisms for improving software reliability. Formal verification is an important part of this effort, since it is the only way to be certain that a given piece of software is free of (certain types of) errors. To date, formal verification has been done manually by specially-trained engineers. Labor costs have heretofore made formal verification too costly to apply beyond small, critical software components. However, if we were able to transform ordinary computer users into ones capable of performing verification tasks, we could achieve a dramatic reduction in the cost of producing verified code. The goal of CSFV (Crowd-Sourced Formal Verification) was to make verification more cost-effective by reducing the skill set required for program verification and increasing the pool of people capable of performing program verification. Our approach was to transform the verification task (a program and a goal property) into a visual puzzle task -- a game -- that is solved by non-experts. The solution of the puzzle is then translated back into a proof of correctness.

Over the course of the program, we developed and improved the code verification tools from the University of Washington Computer Science and Engineering's Programming Languages and Software Engineering group (PLSE), and combined them with games designed and implemented by the UW's Center for Game Science (CGS). We designed a series of games successively titled *Pipe Jam*, *Traffic Jam*, *Flow Jam*, and *Paradox* that present real verification problems as puzzles to players with no technical background.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Verification Approach

Our verification approach is based on type theory. To verify a security property, the types in a program must satisfy certain type constraints. As a simple example, if the program contains the assignment statement "x = y", then the type of x must be a supertype of the type of y. Therefore a proof of correctness can be thought of as a set of constraints involving the statements of the program.

We created a series of games, successively titled *Pipe Jam*, *Traffic Jam*, *Flow Jam*, and *Paradox*, in order to present puzzles to players with no technical background. A game level in these games can also be thought of as a set of constraints that a player is trying to solve. Like many puzzle games, in order to complete a game level in this family of games, the player must find consistent settings for all the game elements.

Because both the games and type-checking are based on constraints, it is possible to create a game level that corresponds to a given piece of code. Specifically, our type analysis system takes as input a Java program and a security property, and it generates as output a set of type constraints that the games present to players as a puzzle to solve. When a player adjusts a game element, this corresponds to selecting a different type for a variable. Because the actual type system constraints are displayed as simple game mechanics, players can help perform verification tasks without needing any prior knowledge of software verification.

If the player is able to solve a given level, the player has also generated a proof that the input piece of code is free from vulnerabilities for the given security property. If the level cannot be fully solved, the constraint graph must contain certain inconsistencies that correspond to type-checking errors for the program -- potential security vulnerabilities that can be examined by a verification expert.

3.2 Game Design

This section introduces and discusses the design of the games *Pipe Jam*, *Traffic Jam*, *Flow Jam*, and *Paradox*.

Iterative Game Design History. *Pipe Jam* was the first game we developed. In *Pipe Jam*, network of pipes in the game are directly generated from the flow (similar to dataflow) properties of a program. The pipes represent program variables, their widths represent types, and the balls represent approximations to run-time values. Player settings for the pipes' widths directly correspond to type annotations in the program that can be mechanically checked and provide a proof of partial correctness.

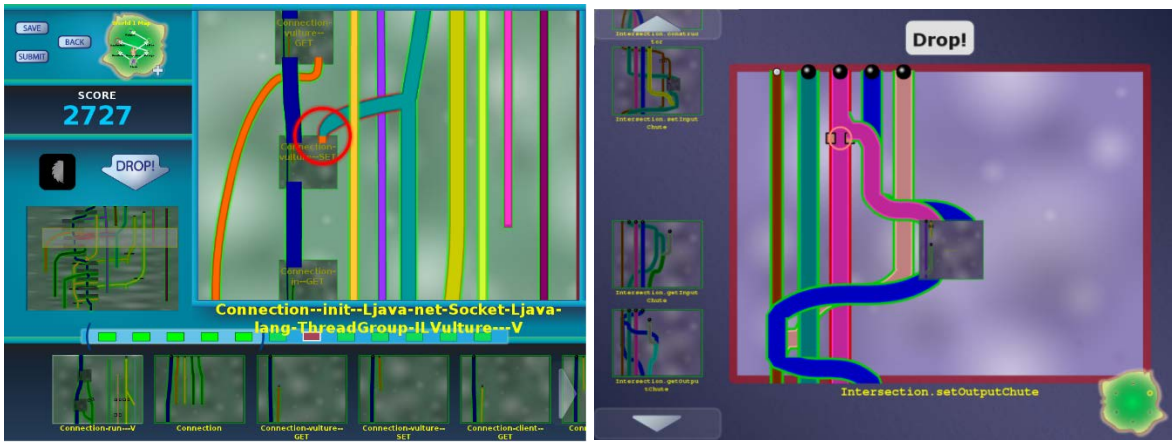


Figure 1. Pipe Jam. Although Pipe Jam successfully represented type information in a game format, there were problems in its representation that stopped it from being fun. This was a problem as we imagined players intrinsically motivated to play the game. Pipe Jam could be confusing for several reasons: colored pipes were difficult to differentiate as the number of colors grew, the pipes were linked across large maps, different uses of the same variable were represented as new pipes.

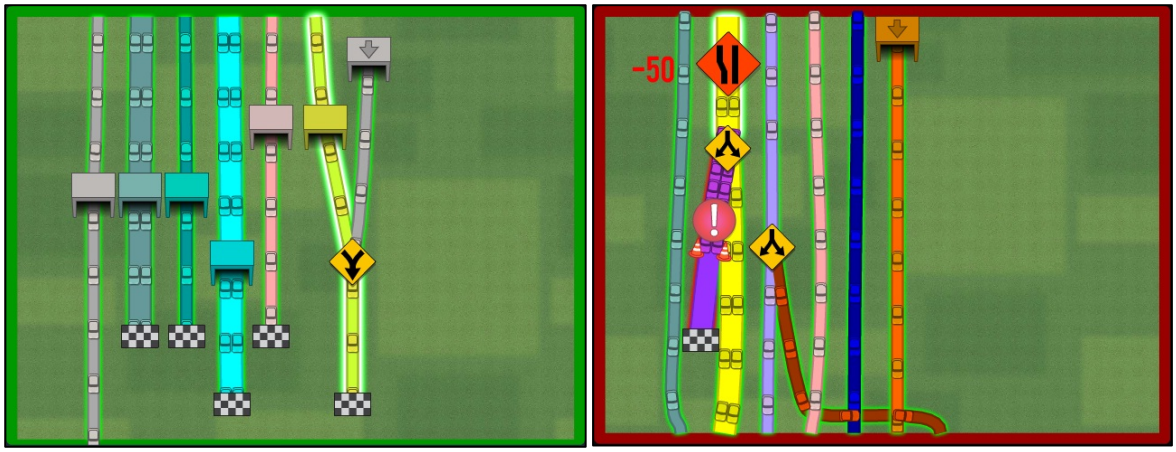
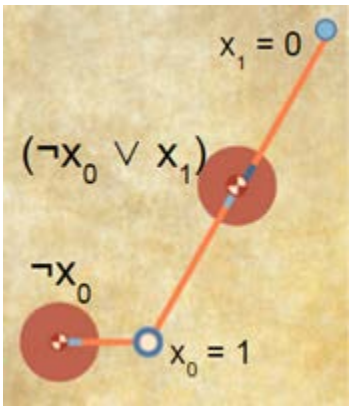


Figure 2. Traffic Jam. The next iteration of the game was called *Traffic Jam*. *Traffic Jam* changed the visual metaphor from abstract pipes with rolling balls to continuously-flowing traffic. This helped players trace pipes that were connected across worlds and gave players immediate feedback when one change created a conflict. The traffic theme was intended to give *Traffic Jam* further appeal by grounding its abstract gameplay in problems recognizable from the real world.

Figure 3. Flow Jam. The next iteration of this approach was called *Flow Jam*. *Flow Jam* attempted to address the problems of the previous two iterations by moving back to a simpler abstract representation that could handle very large levels.

In Phase 2 of the CSFV program, we developed an entirely new game based on what we had learned. This new game was called *Paradox*. *Paradox* addressed many of the problems that were present in the "jam" games by representing levels with a clean and appealing visualization that can be scaled to display levels of enormous complexity.



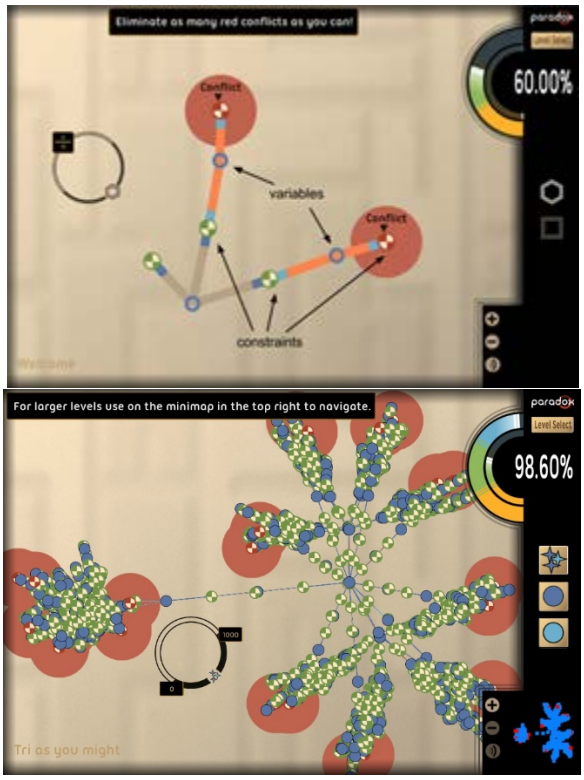


Figure 4. Paradox. A *Paradox* level's elements represent variables and constraints from the underlying constraint problem. A variable node is either light blue or dark blue, representing type qualifiers or their absence in the code being verified. A constraint node requires that at least one of the connected variables has a certain value. If none of the variables for a given constraint are the correct value, then the constraint is marked as a conflict. Edges are the connections between a variable and a constraint when a constraint contains a given variable.

To the left is a *Paradox* level representing the formula:
¬x0 ∧ (¬x0∨ x1). The red circles represent conflicts are shown for the unsatisfied constraints involving variables x0 and x1.

In *Paradox*, the player's goal is to find a setting for the variables that minimizes the number of conflicts. Currently, we represent the variables as boolean values and the constraints as disjunctions over variables or their negations, making the problem the players are solving a maximum satisfiability problem (MAX-SAT).

Maximizing Human Contribution. In order to maximize the contribution that untrained human players of *Paradox* can make to the verification process, players should focus on the portion of problem that is least solvable by automated methods. Up to a certain size, constraint graphs can be solved rapidly by automated solvers and are not challenging for human players. Very large constraint graphs, however -- corresponding to real-world programs such as Hadoop -- can be difficult to understand and present multiple problems for user interface design. Our previous games required players to toggle variables (then called "widgets") individually, which did not scale well to larger levels where humans were most needed.

To address this, *Paradox* provides a "paintbrush" mechanism that allows the player to select arbitrary groups of variables. The player can change them all at once, or the computer can automatically solve them (for groups up to a predetermined limit). Different paintbrushes can allow the player to apply different automated algorithms to their selection. Thus, the main feature of Paradox gameplay is the player guiding the automated methods: deciding which areas of the graph to solve and in what order. Currently players have a-cess to four paintbrushes that have the following effects on the selected variables: set to true, set to false, launch an exact DPLL (Davis-Putnam-Logemann-Loveland) optimization or launch a heuristic GSAT (Greedy procedure for solving SATisfiability problems) optimization. New optimization algorithms can be added to the game as additional paintbrushes.

Additionally, in Paradox, human players are never given small optimization problems (for example, toggling the values of 50 variables to get the optimal score) since automated methods can solve that scale of problem. Instead, they are consistently provided with large and challenging problems that are computationally intractable to solve in an automated manner.

Maintaining Player Interest. In a normal game, levels are created by a game designer with the aim of creating a fun and engaging experience for players. In a formal verification game, however, the levels that are most valuable for players to solve are those generated from the code that is being verified. Since the code in question was most likely created for a very different purpose than making an interesting game level, sometimes levels contain oddities such as enormous sections that are not integral to the solution. Worse, some levels are very large but consist only of repeating structures, resulting in puzzles that are not interesting or challenging for human players.

To study player preferences, a comparable batch of levels was synthesized -- that is, generated randomly and not based on real-world Java code. Using *Flow Jam,* real versus synthesized levels were compared by surveying players to see which type of levels were found enjoyable. Synthesized levels designed to maximize complexity were clearly preferred, with an average 65% preference rating, over real levels, which averaged a 30% preference rating. Although not a rigorous comparison, this indicates that there is room for improving levels generated from real code. We do not yet know whether this preference for synthesized levels in *Flow Jam* carries over to levels in *Paradox*.

To ensure that levels generated from real-world code are interesting enough to entice non-expert human players to solve them, our system adjusts the constraint graphs before they are served to players. For example, irrelevant parts are removed, and a level is broken down into independent levels when possible. If a level can be automatically solved, then it is never given to human players. Subparts of a level may be solved before the player ever sees it. We plan to perform a study comparing levels directly from Java code to levels optimized for human engagement.

Solution Submission and Sharing. Game players on the Internet are not obligated to persist in playing until a level is solved. We found that many players of *Flow Jam* would make some amount of progress, but very few of them would follow through and submit or share their results. Before changing our submission process, there were only about 3,300 submissions compared to about 100,000 levels played (note that players could make multiple submissions on an individual level if desired). Players would often quit midway through without returning to their current state, or fail to notice the level submission/sharing functionality even though they were making progress on the levels.

To address this, *Paradox* automatically submits level configurations to a central server whenever the player's score increases. This takes the burden off of players to manually submit their solutions for evaluation. By adding these submissions back into the system as new level starting points, it also allows future players of a given level to begin with the progress that prior players have made, without requiring them to proactively share solutions with each other.

Sense of Purpose. Another aspect of working with a human population of solvers is motivation. Playtesting has shown that, if players do not understand what they are doing and why they are doing it, they quickly lose interest in the task. In early versions of *Paradox*, players were given the optimizer brush and tasked with painting around conflicts to solve them, leaving them with no sense of what they were actually doing to solve the levels. To fix this, the tutorial now includes a few levels where players must change variables manually. Playtest feedback indicates a much better understanding of the underlying problem and a general sense of purpose when players are required to adjust individual variables in tutorials before using optimizer brushes.

3.3 Verification Tool Improvements

As part of the program we made many enhancements to our verification toolsuite. The most significant of these is handling of Java generics (parametric polymorphism or type variables). This is universally disparaged as the most confusing part of the Java language, and our problems were exacerbated by our need to build upon the existing javac implementation, for tool compatibility. We improved handling of generics for both type checking and type inference.

To handle generics for type inference, we needed to introduce a new type of constraint. Ordinarily, if a type system has two qualifiers (say, @Nullable and @NonNull), then every type in the program can be annotated as either @Nullable or @NonNull. In practice, defaulting and intraprocedural type inference would eliminate the need for many of those annotations, so the program would not be so cluttered; but the effect would be the same if every annotation were explicitly written.

With type variables, this is no longer the case, because writing no annotation is different than writing any specific annotation. Consider the following code:

```
class MyClass<T> {
  @Nullable T field1;
  @NonNull T field2;
  T field3;
}

MyClass can be instantiated as either of
  MyClass<@Nullable String> x;
  MyClass<@NonNull String> y;
```

Figure 5. MyClass

For both types of instantiation, field1's type is @Nullable String; that is, both x.field1 and y.field1 have type @Nullable String. Likewise, both x.field2 and y.field2 have type @NonNull String. But

the type of field3 depends on the instantiation: x.field3 has type @Nullable String, and y.field3 has type @NonNull String.

To accommodate this, we created a new type of constraint variable. An ordinary constraint variable represents a location in the source code and its values are possible annotations. When the constraints are solved, the value is inserted at the appropriate source code location. For a type parameter, the value can be empty. When two variables interact, such as an assignment "x = y;", then the type of the right-hand side must be a subtype of the type of the left-hand side. When one of the variables has a type parameter as its declared variable (such as field3 above), then the constraint may be of two types: if the constraint variable is set to empty, then the constraint is against the upper bound, and otherwise it is normal.

In addition to handling generics in type-checking and type inference, we upgraded our tools to handle Java 8 features such as lambdas (anonymous functions) and method references.

When inserting annotations in source code, we now do so only when the value would be different than the default. This reduces clutter in the source code.

We designed a new qualifier polymorphism mechanism, which allows separate specification of type and qualifier polymorphism. This permits more flexible specification without changing the Java code. This simplifies creating type-checkers, as it permits operation at a higher level of abstraction: qualifiers (semantics) vs. annotations (syntax). This work was motivated by our discovery an unsoundness for any type system that contain all of transitivity, subtyping, and mutability; this had not been described in the literature before. Use of qualifier parameters offers a way to avoid the problem.

We made our architecture pluggable, so that constraints can be solved either via crowd-sourcing or by dispatching to a (satisfiability) SAT solver such as SAT4j, a Java library for solving Boolean satisfaction problems. This enables us to compare human solutions with automated ones, and it enables us to iterate quickly without waiting for gameplay to produce solutions.

We made framework enhancements that make every type checker more accurate:

- The Constant Value Checker performs constant propagation and more: it evaluates side-effect-free methods, tracks array lengths, and yields a set of values rather than just one.
- The Reflection Checker statically resolves 96% of uses of reflection, so that it is no longer necessary to use conservative overapproximations at reflective call sites.
- Field type inference performs intra-class and whole-program analysis to infer field types, which can be used on a subsequent iteration.

We also designed and implemented support for partially-annotated libraries and safe defaults for unannotated code.

4 RESULTS AND DISCUSSION

4.1 Play Statistics

Since the public launch of the combined verigames.com portal in December 2013, over 6,000 unique players have played *Flow Jam* for a combined total of over 7,500 hours of play and over 34,000 level submissions. Since the launch of *Paradox* in May 2015, over 4,500 play sessions have occurred with nearly 1,400 unique players and 16,200 unique level solutions submitted. There are fewer players for *Paradox* because it has not been available as long as *Flow Jam*.

4.2 Proofs of Correctness

Throughout the course of the project, we performed four proofs of correctness, with increasing levels of automation:

1. We proved that the Hadoop-common program (100K non-comment, non-blank LOC) has no operating system command injection attacks. In other words, untrusted data, such as from user input, is never used as part of an operating system command without being properly quoted. If such data were used without quoting, then it would be possible for an attacker to execute arbitrary commands. This proof corresponds to CWE-78: Improper Neutralization of Special Elements used in an OS Command ('Operating System Command Injection').

The proof consisted of annotating and type-checking Hadoop-common. We manually annotated the program with @OsTrusted type qualifiers, wherever data was trusted to be used in an operating system command. All other data is untrusted. We also annotated library routines to indicate whether they produce or require trusted or untrusted data. When the program type-checked, that indicated that the security property holds: untrusted data never flowed to library routines that require trusted data, such as the routines that execute operating system commands

We performed this proof manually. It gave the first indication that our underlying technical approach is feasible and scalable. It also found 5 bugs in Hadoop: locations where validation calls were missing.

2. We proved that Hadoop-common uses format strings correctly, as in printf("%s %d", "a string", 42). This was a manual proof, like the one above. We found and reported an error. This shows that our approach generalizes to multiple type systems.

3. We proved that Hadoop-common does not violate its documented locking conventions. Programmers can document a locking discipline, which indicates which locks must be held in order to access which data. If the program violates its locking discipline, then a race condition occurs, which can corrupt data structures and/or cause inconsistent views even of uncorrupted data.

Our tools inferred a locking discipline for Hadoop and proved that Hadoop satisfied the locking discipline. The annotations are of the form @GuardedBy("lockname"), indicating that data of the annotated type can only be accessed when holding the lock named lockname, and @Holding("lockname"), indicating that lockname is currently held.

This result required no human intervention -- it was fully automated. However, it did not involve the game. This was the first validation of our inference tools, since the previous step had only exercised type-checking and not inference.

4. We re-proved that the Hadoop-common program (100K LOC, lines of code) has no operating system command injection attacks. This proof was different in several important respects. One is that we had corrected many bugs, eliminating the need for human intervention and verification in certain places. Another is that we performed inference rather than mere type-checking, meaning that programmers didn't have to write the annotations. And most importantly, the inference was done by game players. This was the first example of an automated proof: a program was automatically converted to a game, players played the game, completed game levels were converted into program annotations, and the annotations were verified, showing that the program contains no bugs (of one specific variety).

We performed an experiment to compare the cost of two techniques for verifying a program's correctness. One technique is the traditional one in which a human verification expert writes the specifications and then those specifications are automatically verified. The other technique is our crowd-sourced workflow where the specifications are inferred via gameplay, tweaked as needed by the verification expert, and then automatically verified. Our goal is to reduce the overall cost rather than to completely eliminate the human expert's job, a goal we believe is impractical at the current state of the art.

More specifically, our experiment had two developers annotate a program until the program could be automatically verified. One programmer starts from unannotated source code, and the other starts from game results (inference).

We verified the Nexus-SS program, which is a source code repository system for the Maven build tool; it consists of 46224 non-comment, non-blank LOC. We verified that it does not use hard-coded credentials (CWE-798, https://cwe.mitre.org/data/definitions/798.html), where CWE stands for Common Weakness Enumeration. This vulnerability exists when “The software contains hard-coded credentials, such as a password or cryptographic key, which it uses for its own inbound authentication, outbound communication to external components, or encryption of internal data.”

The type system used for verification consists of two types: @HardCoded indicates that a value is found in the program source code or computed only from values found in the program source code, and @NotHardCoded is all other values. Interestingly hard-coded values are trusted in other contexts, such as assuming that they do not contain data that would be an injection attack.

The results are presented in Figure 5.

Starting condition	Total time	Machine time	Manual time
Unannotated	45	7	38
Game results	4	3	1

Figure 6. Results. Times are in minutes. The "Machine time" column is type-checking time, which is human wait time.

These timings do not include annotating libraries, such as which ones require non-hard-coded credentials (this determines the proof goal and is required in both cases), or game play (crowd time, machine time to generate boards). In this particular case, the game computed 23 correct annotations, and the human merely verified them, which took little time. We would like to re-run this experiment with more and bigger programs and with different type systems.

4.3 Free-to-play vs. Paid Players Analysis

Testing was performed to determine the feasibility of hosting the game (*Paradox*) on Amazon Mechanical Turk, a crowdsourcing marketplace where users are paid based on predefined criteria to play the game. After some trial and error to determine reasonable rewards, we implemented a policy where users were paid one cent for completing a shortened set of tutorials. If the tutorials were completed, users were granted qualifications allowing them to complete game levels generated from actual code. These actual game levels paid a flat payment of ten cents for each user per level, with a bonus of 25 cents for each period of 5 minutes where the user was actively increasing their score (up to a maximum bonus of $1.25). Identical levels were given to players on verigames.com and submitted as paid tasks for Amazon Turk users.

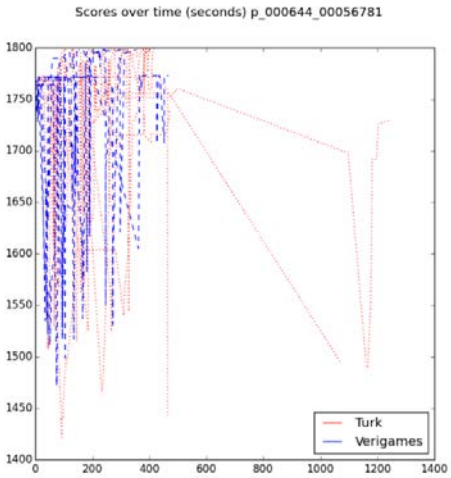


Figure 7. Time to Completion. The figure below left illustrates the trend that verigames.com players tend to reach the highest score more quickly within a given play session than Mechanical Turk users. The y-axis shows scores over time for each player, the x-axis corresponds to the number of seconds since the beginning of each play session with all players' play sessions for this particular level (p_005204_v522558) overlaid on top of each other. However, in terms of the overall number of solutions being generated since the time that the levels are released, the Mechanical Turk group tends to have tens of solutions within 24-48 hours compared to the smaller verigames.com group. Below right is a figure showing the top scores per play session and the number of days since the release of the level that it took to receive those scores per group.

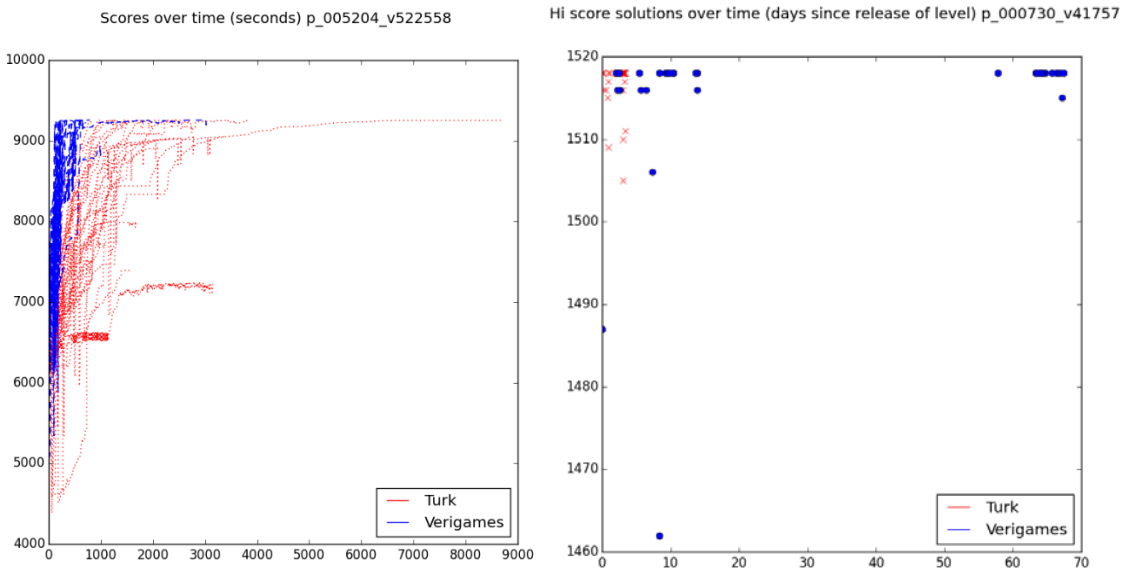


Figure 8. Willingness to Enter Suboptimal Solution Space. One recurring issue with crowdsourced efforts is that players can confine themselves to local maxima, where users attempt to monotonically increase their score instead of pursuing a strategy that may temporarily lower their score in order to find the best configuration. This is undesirable because

it restricts the overall solution space. One might expect that paid players value their time more and are less willing to undo progress. However, we did not observe this with Mechanical Turk Players. In general, approximately the same portion of players in Mechanical Turk and verigames.com pursued lower score strategies. The figure to the left shows one level which illustrates the presence of those strategies where the starting score is 1750 and players of both groups are observed to go below that score in order to pursue new solutions.

Solving Strategies. Both the verigames.com group and the Mechanical Turk group were observed to have used all three brushes, with a heavy emphasis on the Optimizer Brush. There was a slightly more pronounced bias towards the use of the Optimizer brush in the verigames.com group compared to the Mechanical Turk group as shown in the figures below, where paint actions are shown in yellow for the verigames.com group and the Mechanical Turk group, respectively.

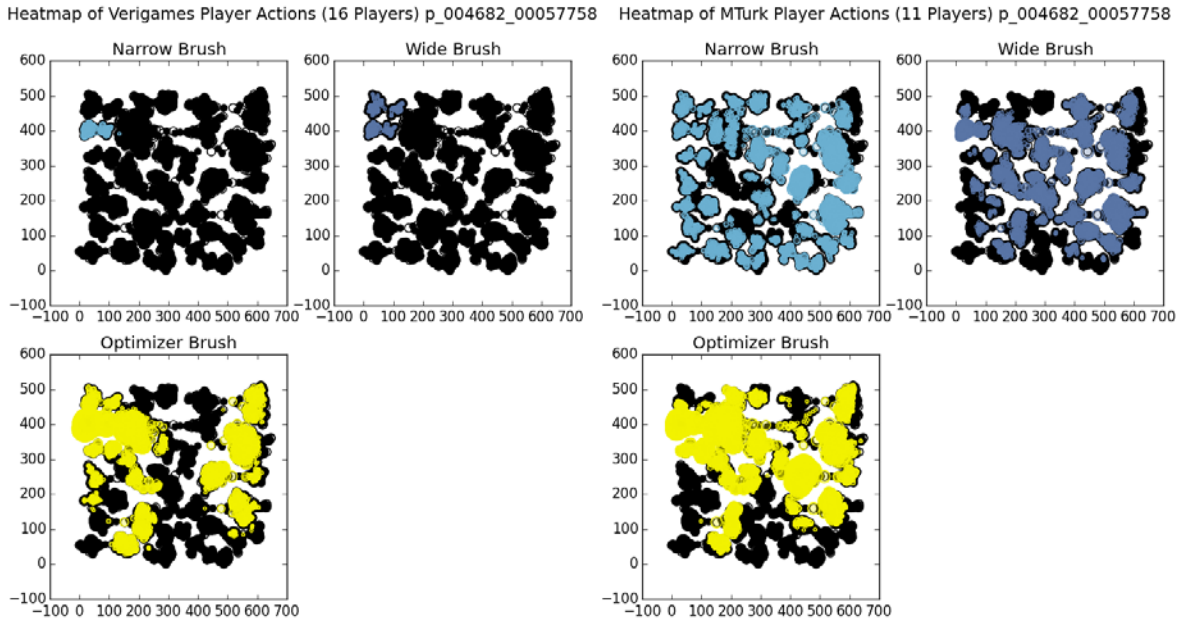


Figure 9. Player Solutions. Generally the high scores for each level contained an even split between verigames.com and Mechanical Turk players. In addition, the solutions themselves do not appear to correlate amongst members of the same group. In other words, there do not appear to be any distinguishing characteristics unique to the verigames.com group solutions compared to the Mechanical Turk group solutions. The figure below illustrates the difference in solutions for a given level, with the number of wide/narrow variables in each shade of blue.

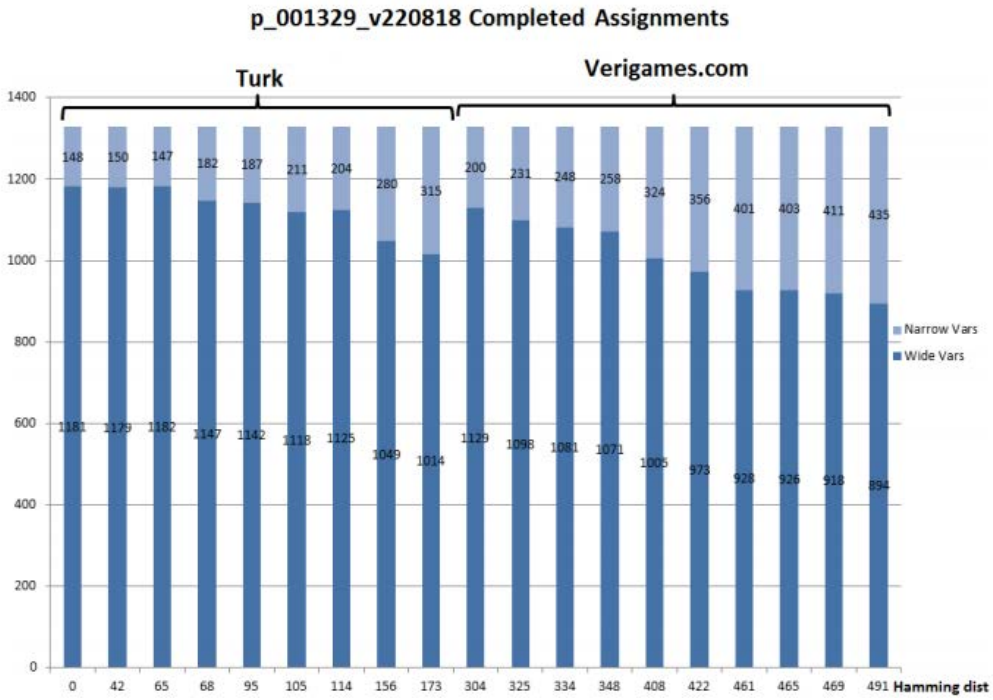


Figure 10. Free-to-play vs. Paid Players Conclusions. With our current rewards scheme, the Mechanical Turk group can return 10-20 solutions for a given level within 48 hours. To compare, due to low traffic on the verigames.com site, each level takes a few weeks to gather the same volume of solutions. Given the data above and the fact that the solutions appear to be equally desirable from both groups, the Mechanical Turk methods appear to be preferable.

5 CONCLUSIONS

We attempted a radical new approach to program verification by transforming the problem domain into one in which non-experts could interact with a program through the framework of a video game. We developed and improved important code verification tools and combined them with a series of game interfaces that anyone could play. Through constant playtesting we eventually designed a game that

We made a number of concrete proofs on Hadoop, a large piece of a widespread and currently-used program in support of developing and testing our approach, and performed an experiment to compare the cost of two techniques for verifying a program's correctness which shows the CSFV approach can dramatically reduce the time required by an expert, and therefore the overall cost, of formal verification.

5.1 Recommendations

The lessons that have guided development from the earlier game *Flow Jam* to the current game *Paradox* naturally point towards future areas of study. These topics include player performance versus fully automated solvers, player effectiveness with different graph representations and groupings, and differences between volunteer players and compensated players. Also, given its general nature, problems from other domains that can be encoded as maximum satisfiability problems (MAX-SAT) could be used to create levels in *Paradox*. Our game design may also extend to other types of constraint satisfaction problems that can be visualized as a factor graph.

6 REFERENCES

1. "Verification games: Making verification fun" by Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Nathaniel Mote, Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popovic. In *FTfJP'2012: 14th Workshop on Formal Techniques for Java-like Programs*, (Beijing, China), June 12, 2012.
2. "A type system for regular expressions" by Eric Spishak, Werner Dietl, and Michael D. Ernst. In *FTfJP'2012: 14th Workshop on Formal Techniques for Java-like Programs*, (Beijing, China), June 12, 2012.
3. "Type Annotations specification (JSR 308)" by Michael D. Ernst. Oct. 2011.
4. "Building and using pluggable type-checkers" by Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering,* (Waikiki, Hawaii, USA), May 25-27, 2011, pp. 681-690.
5. "Lessons learned in game development for crowdsourced software formal verification." Drew Dean et al. "Solution 2: Flow Jam and Paradox." Tim Pavlik, Craig Conner, Jonathan Burke, Matthew Burns, Werner Dietl, Seth Cooper, Michael D. Ernst, and Zoran Popović. *USENIX Summit on Gaming, Games, and Gamification in Security Education* (2015).

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

CGS	Center for Game Science
CSFV	Crowd-Sourced Formal Verification
CWE	Common Weakness Enumeration
DPLL	Davis-Putnam-Logemann-Loveland (algorithm for satisfiability)
GSAT	Greedy procedure for solving SATisfiability problems
LOC	lines of code
MAX-SAT	maximum satisfiability
OS	operating system
PLSE	Programming Languages and Software Engineering
SAT	satisfiability
SAT4j	a Java library for solving Boolean satisfaction problems